Evolutionary Testing of Autonomous Software Agents

Cu D. Nguyen, Anna Perini,
and Paolo Tonella
Fondazione Bruno Kessler
Via Sommarive, 18
38050 Trento, Italy
{cunduy,perini,tonella}@fbk.eu

Simon Miles, Mark Harman,
and Michael Luck
Department of Computer Science
King's College London
Strand, London WC2R 2LS, UK
{simon.miles,mark.harman,
michael.luck}@kcl.ac.uk

ABSTRACT

A system built in terms of autonomous agents may require even greater correctness assurance than one which is merely reacting to the immediate control of its users. Agents make substantial decisions for themselves, so thorough testing is an important consideration. However, autonomy also makes testing harder; by their nature, autonomous agents may react in different ways to the same inputs over time, because, for instance they have changeable goals and knowledge. For this reason, we argue that testing of autonomous agents requires a procedure that caters for a wide range of test case contexts, and that can search for the most demanding of these test cases, even when they are not apparent to the agents' developers. In this paper, we address this problem, introducing and evaluating an approach to testing autonomous agents that uses evolutionary optimization to generate demanding test cases.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Reliability, Verification

Keywords

Evolutionary Testing, Autonomous Agents

1. INTRODUCTION

The concept of autonomy, *auto = self*; *nomos = law*, refers to self-governance, freedom from external influence and/or authority. It is key to making software agents a distinctive class of computational systems and is an enabling technology for building open, dynamic, and complex systems. Concerns about autonomy have appeared since the advent of artificial intelligence, because intelligent entities should be able, at least to some degree, to autonomously decide which actions to take. More recently, however, several different aspects related to agent autonomy, such as operationalization, architecture, autonomy adjustment, and others, have been discussed with greater intensity [11].

Now, as software agents with built-in autonomy are increasingly taking over control and management activities, such as in automated vehicles or e-commerce systems, testing these systems to make sure that they behave properly becomes crucial. Testing traditional software systems, which have reactive (or input-output) style behaviour, is known to be non-trivial, but testing autonomous agents is even more challenging, because they have their own reasons for engaging in proactive behaviour that might differ from a user's concrete expectations, yet still appropriate; the same test input can give different results for different executions.

To illustrate the issues involved in testing autonomous as opposed to non-autonomous components, consider the following example. A component providing compression functionality is tested according to two criteria: *(1)* the output of the compress function is smaller than the input; *(2)* the output of the compress function, when given as input to the decompress function, returns the original input. In this simple non-autonomous testing example, the tester may not know the compression algorithm used by the component, and does not know the exact data that will be output from the compress function. However, it can be expected that the output will remain the same for a given input over time; that is, *the run-time context in which the tests are performed does not change the test outcome.* In comparison, the same tests can be applied by a client asking an *autonomous* agent to compress data, but there may be run-time variation in the actual behaviour of the agent. The agent may have its own internal goals and knowledge, both of which may change over time, and these can affect the result returned, if any. For example, the agent could choose different compression algorithms based on the resources available, or delegate the task to different subordinates over time, depending on which it currently considers to provide the best service. To properly test the autonomous agent thus requires more than the single tests on the predictable component: it requires the same tests to be applied to the agent in a range of contexts. Ensuring that the range of contexts (in this example, the range of algorithms or subordinates available and chosen) tested against is adequate to declare the agent correctly functioning is, therefore, a hard but important task.

Existing work on agent testing (for example, [1, 17]), deals mainly with agent interactions and constraint enforcement. In [1], agent interactions are observed to detect interaction problems such as missed communication. By contrast, Rodrigues et al. [17] propose to exploit social conventions (i.e., norms or rules that prescribe permissions, obligations, and/or prohibitions of agents) in an open multiagent system

for integration testing. During test execution these constraints will be checked, and violations reported.

In the context of agent autonomy, however, only looking for violations (related to constraints, norms, and the like) is seldom sufficient. Confidence in the autonomous behaviours of an agent must be established. In other words, we need to answer the question: *are these autonomous agents dependable?* In addition, autonomous agents should be able to operate in open environments where different and even unpredictable circumstances may arise. Testing must also deal with this aspect.

In response to these concerns, in this paper we specify an evolutionary testing approach, guided by a stakeholder's quality criteria, to test autonomous agents. We describe the general testing procedure, evaluate it with a case study, and analyse the benefit it brings. Our approach can be outlined as follows. Stakeholder *requirements related to autonomy* (i.e. requirements that an autonomous agent may satisfy differently depending on its context) are transformed into *quality functions*. We then evolve increasingly demanding test cases using the quality functions as fitness measures, where the lower the quality the agent produces, the tougher we can infer the test case is, and the more likely the test case is to survive and reproduce as the evolution progresses. By evolving steadily tougher test cases, we test the agent in a range of contexts, including those in which it is most vulnerable to poor performance.

The motivations behind our approach are twofold. Firstly, we believe that quality functions –derived from requirements related to autonomy– can be used to evaluate autonomous agents to build confidence in their behaviours, because meeting such requirements contributes to agent dependability. It is worth noticing that our aim is to evaluate the exhibited performance of autonomous agents, not the mechanism underlying autonomy itself. Secondly, because it is automated, the use of evolutionary algorithms can result in more thorough testing at comparatively lower cost than other forms of testing, such as manual test case generation, which is tiresome, error-prone and expensive. That is, large number of challenging circumstances, generated by evolution, can be used to test agents, with each test case seeking to expose any possible faulty behaviour.

Since autonomous agents can behave differently under the same setting, statistical evaluation methods may be needed to properly evaluate each test case; for example, when the expected quality must be achieved with a high average probability.

We implemented and applied the proposed approach to a case study: a simulation of a cleaning agent. The obtained results show that our approach outperforms random testing, the basic alternative technique available.

The remainder of the paper is structured as follows. Section 2 gives background notions about evolutionary testing and presents related work. Section 3 and 4 introduce our approach, while Section 5 discusses the case study used to illustrate the approach and to assess its effectiveness. Finally, Section 6 concludes this work and presents future improvements.

2. BACKGROUND AND RELATED WORK

2.1 Background

Evolutionary testing (ET) [13, 21] is inspired by the evolution theory in biology that emphasizes natural selection, inheritance, and variability. Individuals that are fitter have a higher chance of surviving and producing offspring with favourable characteristics of individuals being inherited. In ET, we usually encode each test case as an individual in a population of candidate test cases. In order to guide the evolution towards better test suites, a fitness measure is defined as a heuristic approximation of the distance from achieving the testing goal (e.g., covering all statements or all branches in the program). Test cases with better fitness values have a higher chance of being selected for survival and reproduction. Moreover, mutation is applied during reproduction, thereby enhancing diversity; an important consideration in any and all evolutionary processes.

The key step in ET is the transformation from testing objective to search problem, specifically through the fitness measure; different testing objectives give rise to different fitness definitions. For example, if the testing objective is to exercise code inside an *if* block, one can define a fitness function that gives lower values (considered as better) to test cases that are closer to making the conditions of the *if* statement true; the lowest (best) value is given to the test cases that make the conditions true, so that the code inside the *if* block is executed. Once a fitness measure has been defined, different optimization search techniques, such as *local search, genetic algorithm, particle swarm* [13] can be used to generate test cases aimed at optimizing the fitness measure.

2.2 Related work

The use of evolutionary search techniques for the automatic generation of test data has been receiving increasing interest from many researchers, reflecting a growing trend towards Search Based Software Engineering (SBSE) [10]. The SBSE approach is also bridging the technology transfer divide to industry. For example, Bühler and Wegener [4] applied evolutionary functional testing in two automatic systems: automatic parking and brake assistant systems for Daimler's Mercedes class cars. The systems investigated are automatic, not autonomous, but the results of evolutionary functional testing, which outperforms random and manual testing, show the potential of this technique.

More directly relevant to this work, Nguyen et al. [14] describe the combination of evolutionary and mutation testing for testing autonomous software agents. For Nguyen et al., fitness is defined to be mutant score, i.e. the number of mutants killed. A mutant is a modified version of the original agent under test containing a single deliberately seeded fault. A mutant is said to be killed by a test input if the input causes the mutant to exhibit different behaviour to the original. Nguyen et al. approach the problem of testing for autonomy indirectly, by using constraints and the fact that while software agents are free to evolve, their behaviours must obey the norms and rules that govern the operation of the system in which agents are situated, or the constraints imposed on the behaviours. Test cases that kill more mutants are likely to reveal faults in the original agents, hence they have better fitness values.

Fulfilment (or violation) of norms, and satisfaction (or otherwise) of requirements, are directly comparable, as both define what *should* occur. However, both the aims and the approach taken here differ fundamentally from that of norm monitoring. Systems that detect the violation of norms pro-

vide tests (that just happen to occur at run-time) on agents: whenever a norm may be fulfilled or violated, this is determined and reported. However, the norm-monitoring approach does not attempt to rigorously test the agents involved. The only 'tests' performed are due to states of the system that happen to come about; the main concern is to check actual satisfaction of requirements (compliance with norms) at run-time. However, unless rigorous testing has occurred beforehand, the use of this approach alone could lead to a catastrophic violation, which would be detected but only dealt with after it has occurred (when it is too late). Our approach, on the other hand, attempts to cover a range of test cases *in preparation* for the agents under test to operate in a running system. The need for quality and range of test cases is the most significant factor influencing our approach, as opposed to compliance in any single instance.

In the agent-oriented software engineering literature, a variety of research tackling different aspects of agent testing have been proposed. Coelho et al. [5] proposed a framework for unit testing of *MAS* based on the use of *Mock Agents*. Their work focuses on testing the roles of agents at the agent level. Mock agents that simulate real agents in communicating with the agent under test are implemented manually, each corresponding to one agent role. Sharing this inspiration from JUnit [8], Tiryaki et al. [19] proposed a test-driven *MAS* development approach that supports iterative and incremental *MAS* construction. A testing framework called SUnit, which is built on top of JUnit and Seagent [7], was developed to support the approach, allowing the writing of tests for agent behaviours and interactions between agents. Dung et al. [12] proposed a semi-automated process for comprehending software agent behaviours. Their approach imitates what a human user, such as a tester, does in software comprehension: building and refining a knowledge base about the behaviours of agents, and using it to verify and explain behaviours of agents at runtime. The ACLAnalyser [1] tool analyzes runs on the JADE [18] platform, intercepting all messages exchanged among agents and storing them in a relational database. This approach exploits clustering techniques to build agent interaction graphs that support the detection of missed communication between agents that are expected to interact, unbalanced execution configurations, and overhead data exchanged between agents.

As software agent development emerges, testing software agents is receiving increased research attention. The above work focuses on agent interactions, which is reasonable, because agents communicate primarily through message passing. However, none of this previous work explicitly tackles agent autonomy, which is the objective of this paper.

Núñez et al. [15] introduced a formal framework to specify the behaviour of autonomous e-commerce agents. The desired behaviours of the agents under test are specified by means of a formalism, called *utility state machine*, that embodies users' preferences in its states. The operational traces of the agents under test are checked against these specifications in order to detect problems. Our work differs from [15] in that we investigate how to generate effective test cases based on the exhibited performance of the agents under test, not on their specifications.

3. OUR APPROACH TO AGENT TESTING

Autonomous software agents differ from traditional soft-

ware in that they have their own goals and operate in a self-motivated fashion.

We propose to apply the recruitment metaphor to evaluate autonomous software agents. Here, software agents are candidates and stakeholder requirements are used as evaluation criteria. Each agent is given a trial period in which a number of tests with different difficulty levels are to be solved. Agents are recruited (trusted) only when they pass the required quality criteria.

In requirements engineering, the importance of application domain stakeholder goals has long been recognized. As such, the concept of *goal* has been considered central to a number of goal-oriented requirements engineering (GORE) approaches [3, 6]. In GORE, soft-goals play a key role in representing non-functional or "ility" requirements, such as dependability, availability, and so forth, which denote the important criteria for evaluating autonomy. Returning to the recruitment approach to evaluating autonomous agents, we propose to use stakeholder soft-goals as criteria for assessing the quality of autonomous agents, since satisfying quality criteria derived from these soft-goals is likely to indicate that the agents are reliable.

We propose an evaluation methodology consisting of two main steps:

1. *Representing stakeholder soft-goals as quality functions*. Relevant soft-goals that need to be used to evaluate agent autonomy are transformed or represented as quality functions for measuring stakeholder satisfaction. This transformation is domain specific and depends on the nature of the soft-goal as well as on the problem domain.

2. *Evolutionary testing*. In order to generate varied tests with increasing level of difficulty, we advocate the use of meta-heuristic search algorithms that have been used in other work on Search Based Software Engineering [10], and, more specifically, we advocate the use of evolutionary algorithms. The quality functions or thresholds of interest are used as objective functions to guide the search towards generating more challenging test cases.

As an example, Figure 1 illustrates the goals of a specific stakeholder in an airport organization, namely the building manager, who decides to assign the goal of airport cleanliness to a cleaner agent. The notation used in the figure is proposed in Tropos [3]. In this example, the agent must operate autonomously, with no human intervention. On the other hand, the agent must be robust and efficient as stated in the two stakeholder's soft-goals, depicted as two cloud shapes. Applying the proposed approach, these two soft-goals can be used as criteria to evaluate the quality of the cleaner agent: the agent can be built with a given level of autonomy. Robustness and efficiency are two key quality criteria for evaluating it. If the cleaner agent can perform tasks autonomously, but is not robust (for example, it crashes), it is not ready to be deployed.

Regarding the *robustness* soft-goal, two sub-goals decomposed from robustness that are taken into account in this example are *maintaining-battery* and *avoiding-obstacles*. (For brevity, this analysis is not shown in Figure 1.) We can define a threshold for the *maintaining-battery* capability (e.g. 10%), and monitor the battery level at runtime. The battery level must be maintained at a sufficiently high level ($>$
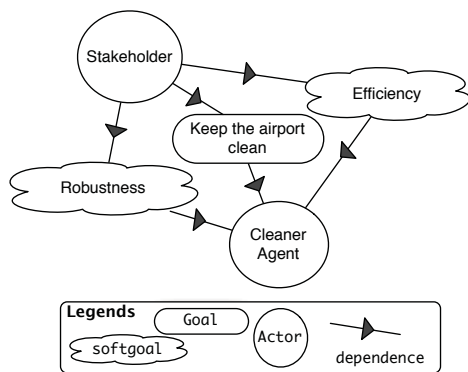
Figure 1: Example of stakeholders' soft-goals

10%) within the period considered. Figure 2 shows an unacceptable scenario in which the battery level drops below 10%.
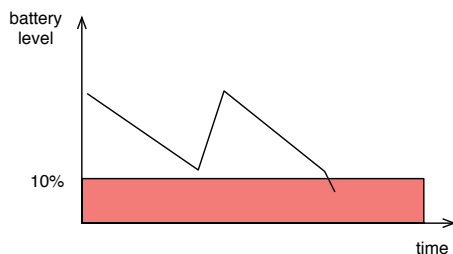

Figure 2: An unaccepted scenario of the battery level

Similarly, for the soft-goal *avoiding-obstacles*, one can define the distance to the closest obstacles during movement as a quality criterion. Correspondingly, a quality threshold ε (distance units) can be defined, and the agent must stay farther from obstacles than this threshold.

In reality, apart from robustness, we can impose many other requirements related to autonomy on the cleaner agent: *stability, efficiency, safety*, for example. Stability demands the agent should avoid dropping its goals too frequently. Efficiency requires the agent to finish cleaning an area after a specific amount of time, or it must bring an amount of waste (e.g. 10 Kg) to the dustbins per hour. The safety requirement demands that the agent must switch to its 'safe mode' in undesirable circumstances, e.g., arms malfunction.

4. EVOLUTIONARY TESTING OF AUTONOMOUS AGENTS

4.1 Encoding test inputs and evaluating results

Let us examine the possible input space for autonomous agents. Georgeff and Ingrand [9] present a minimal design of a reference architecture for BDI agents [2], which is widely applied to build autonomous agents. In the architecture, agents perceive the outside world (from the agents' perspective) through a set of sensors and make changes to the world through a set of effectors. Weyns et al. [22] complement the architectural picture of multiagent systems with a reference model for the environment, in which agents access the environment by employing either perception (sense and percept),

action (make changes to the environment), or communication (send and receive messages). Though the two architectures appear to be slightly mismatched because of the communication element, they actually fit well with one another as agent communication involves environmental facilities — incoming (inbox) or outgoing (outbox) buffers — receiving a message can be seen as perceiving the inbox, and sending one as placing it in the outbox. To this end, the outside world from the perspective of an agent consists of environmental artefacts and other agents. They are considered as test inputs in testing software agents. Autonomous agents monitor these elements actively to detect and reply to events or changes in a timely fashion, or they can receive stimuli from these elements and react proactively.

Encoding test inputs for ET is a crucial step as inappropriate encoding can result in losing inheritance in missed opportunities. We propose to encode each input element by means of one gene. A test case, consisting of a set of all investigated elements, will be encoded as a chromosome. A fundamental requirement for encoding test cases as chromosomes is that offspring obtained by crossover of parental genes contains only slight, non-disruptive changes (combinations) compared to their parents. Since we encode each environmental input by one gene (a string of bits of 0 or 1, for instance), the crossover operation on two genes is expected to produce two new genes that inherit the characteristics of the two original genes. Thus, this 'non-disruptive' requirement is respected.

Since agents of the same kind (or instances of the same agent) might give different outcomes in the same environment, i.e. from the same inputs, evaluating test results requires statistical approaches. That is, test execution for a single input setting may need to be repeated in order to acquire the overall picture; a single execution per test case will seldom provide a thorough test result judgment.

4.2 Testing procedure

The automated testing procedure is presented in Figure 3. Its four steps are described as follows:

1. *Generate initial population.* A set of test cases is called a *population*. Each test case is an individual in the population, and it represents a combination of states (i.e. values) of environmental inputs. The initial population can be generated randomly or taken from existing test cases, created by testers.

2. *Execution and monitoring.* Test execution involves inserting the autonomous agents under test into the testing environment, made up of environmental inputs, so that they can operate (i.e. perform tasks or achieve goals). At the same time, a monitoring mechanism is needed to observe and record the behaviours of the autonomous agents. A number of executions need to be performed repeatedly (or in parallel) in order to provide sufficient data to statistically measure fitness values in the next step.

3. *Collect observed data and calculate fitness values.* Cumulative data from all executions are used to calculate fitness values of selected test cases. The method of calculating fitness values depends on the stakeholder softgoal of interest and the problem domain. Since calculated fitness values provide insight about improvement, if no improvement is observed after a number of gen-

erations, the test procedure stops. Otherwise Step 4 is invoked.

4. *Reproduction.* Two individuals are selected, and then the crossover operation is used to produce one or two new offspring. Finally, mutation is applied with a certain probability on one (or both) offspring. As with natural evolution, selection is biased in favour of fitter individuals.
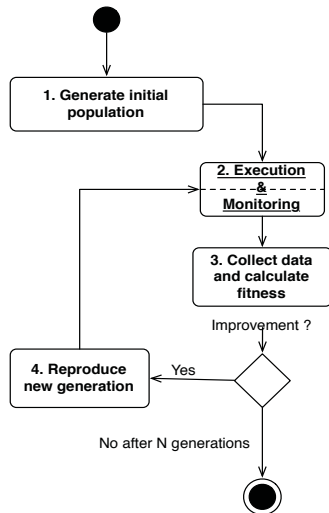

Figure 3: Evolutionary testing procedure

5. CASE STUDY

In this section, we further analyze the cleaner agent, introduced briefly in Section 3, and build a simulation of an agent system composed of an artificial environment and the cleaner agent to evaluate the proposed approach. We describe, in detail, the functionalities of the agent and the way we use soft-goals to guide test generation and ultimately evaluate the quality of the agent.

As mentioned in Section 3, we choose two soft-goals: *robustness* and *efficiency* to evaluate the quality of the cleaner agent. By analyzing *robustness*, two further goals decomposed from it are *maintaining-battery* and *avoiding-obstacles*. Similarly, *efficiency* can be decomposed into sub-goals as well. All of them must be taken into account while evaluating the quality of the cleaner agent. Each soft-goal gives rise to a fitness function that can guide the generation of test cases. In this paper we investigate only the goal *avoiding-obstacles*, using a fitness function derived from the goal to guide the generation of test inputs. The testing objective is to make sure that the agent does not hit any obstacles.

5.1 Application

The artificial environment is a square area, A. In the area A there can be obstacles, dustbins, waste, and charging stations located randomly. We define an *environmental setting* as a particular configuration of A, in which numbers of obstacles, dustbins, waste, and charging stations are located at particular locations. Different settings pose different levels of difficulty in which the cleaner agent must operate.

The cleaner agent is in charge of keeping that area clean. In particular, it needs to perform the following tasks autonomously:

1. Exploratory location of important objects.

2. Look for waste and bring it to the closest bin.
3. Maintain battery life, with sufficient re-charging.
4. Avoid obstacles by changing course when necessary.
5. Exhibit alacrity by finding the shortest path to reach a specific location, while avoiding obstacles on the way.
6. Exhibit safely by stopping gracefully should movement become impossible or battery level too low.

These are all requirements related to autonomy; the way in which the agent achieves them differs depending on the context in which it finds itself. Each functionality gives rise to a goal that the agent needs to achieve or maintain, and multiple goals are active simultaneously during operation. For instance, while exploring the area, the agent needs to avoid obstacles and maintain its battery.

The simulation environment is implemented in JADEX [16], extending an existing example of JADEX with a sophisticated capability to avoid obstacles. The cleaner agent contains a belief base where information about current location, visited locations, obstacles, dustbins, charging stations, and so on are stored. In addition, the agent has a number of goals and associated plans, with goal deliberation based on goal conditions, such as creation, adoption and inhibition conditions. At runtime, goals are adopted autonomously on the basis of goal deliberation.

5.2 Preparation

5.2.1 Encoding test inputs

In this case study, an environmental setting (or a test case) is composed of the quantity and location of obstacles, dustbins, waste, and charging stations. Each of these factors is encoded as a single gene, as follows (See also Figure 4):

- Divide the area A into RxR cells, R is called *resolution*.

- Place objects (i.e. obstacles, waste, bins, charging stations) into cells. A cell containing an object is denoted by 1, while a content–free cell is denoted by 0.

1	0	1	0	1	0
0	1	1	0	1	0
0	1	1	0	1	0
1	1	1	0	1	1
0	1	0	1	0	0
1	0	1	0	0	0

Figure 4: Encoding test inputs

The resolutions of the environmental factors can be different and their quantity can be controlled in evolutionary testing. For instance, we can choose the number of dustbins and charging stations to be as small as or smaller than they are in reality, while the amount of waste and number of obstacles can be chosen to be much higher.

During evolution, genes are crossed over and/or mutated, resulting in new environments that combine previous environments or in which objects change their location. In other

words, the new environments only have relatively slight changes compared to their parents, thus the requirements of encoding test inputs, mentioned in Section 4.1, are respected.

5.2.2 Fitness computation

We define a fitness function f based on the distance to obstacles encountered during the operation of the agent. Real-time observations of the distance of the cleaner agent to all obstacles are performed to measure f. Moreover, since the test outcomes are different even for the same test input, we need to repeat the execution of each test case several times to measure statistical data representing the test outcomes. This section determines a reasonable value for this repetition.

In exactly the same initial environmental setting, different executions can result in different trajectories of the agent. This is due to the random targets that the agent chooses to reach, while exploring the environment. As a result, the agent can find itself in trouble if the randomly-selected target is close to obstacles, or if the path to the target is obstructed by obstacles so that the probability of hitting obstacles becomes high. On the other hand, if by chance, all the selected targets happen to be far away from obstacles, then the probability of hitting obstacles would be low.

In order to find an effective environment where the probability of encountering obstacles is high, we must run each test case a number of times in order to reduce the influence of those non-deterministic factors in agent decision making. In the following, we determine how many executions of each test case is reasonable, and how much time is needed for each run so that the agent has enough time to exhibit its behaviour. For the second question, 40 to 60 seconds is determined to be sufficient for each execution, since within that amount of time, the agent can visit all cells in the testing area several times.

To answer the first question, we randomly generate a number of test cases and execute them repeatedly a number of times. Figures 5(a) and 5(b) show cumulative box-plots of the closest distances to obstacles over the number of executions of two test cases $TC1$ and $TC2$. In each execution we measured the distance of the agent to the closest obstacles in real-time. We use a box-plot presentation because it shows, not only the closest and furthest distance of the whole operation time, but also the 'hardness' of a test case. That is, the quartiles of 25% and 75% of the distances form a range that provides the typical dispersion of distances. If the range is close to 0, the probability of encountering obstacles is high.

In general, we can observe, from Figure 5, that the boxes in each figure tend to converge in terms of size and position. We perform a pilot experiment with a large number of test cases, each has been executed a number of times. The fitness value is non deterministic, because of the non deterministic behaviour of the agent under test. However, the average fitness value converges toward its final value after 4.6 executions (on average), as apparent from the cumulative box plots. Hence, we use 5 test case executions in our experiments to determine the fitness value associated with a test case.

5.2.3 Fitness function

Let D be the vector of all the closest distances to obstacles observed in all executions, and ε be the smallest distance allowed (user-defined threshold). Then, fitness is defined as
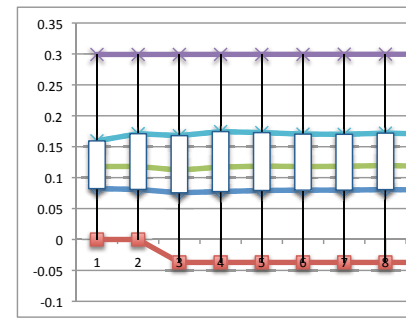
(a) Cumulative box-plots for $TC1$
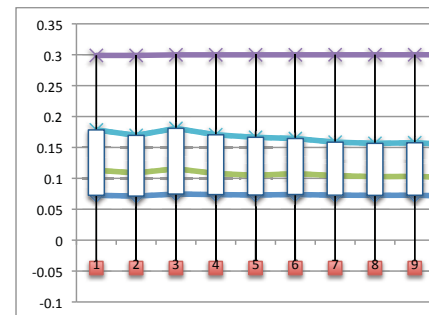
(b) Cumulative box-plots for $TC2$

Figure 5: Cumulative box-plots for two test cases: distance versus number of executions
follows:

$$f = \begin{cases} min(D) + w_1 * quartile1(D) + w_3 * quartile3(D) \\ \qquad \text{if } min(D) > \varepsilon, \\ min(D) - \varepsilon \qquad \text{if } min(D) \leq \varepsilon, \\ +\infty \text{ if the agent cannot move and suspend safely.} \end{cases}$$

where $min(D)$ is the smallest value of the vector D, $quartile1(D)$ is the quartile of 25% of D, and $quartile3(D)$ is the quartile of 75% of D. The weight of the two last values w_1, w_3 must be close to 0 as they are less important than the $min(D)$ value with respect to f. In fact, the reason for using the quartile values is that, among a set of test cases, we favour those that have distance dispersion (i.e. box-plots) close to 0 if they have the same $min(D)$. We have also performed pilot experiments without taking into account the distance dispersion, and the obtained results show that f is less effective when the dispersion is discarded.

The search objective is to bring the box-plots close to the threshold ε whenever $min(D)$ is still greater than the threshold ε. Otherwise, only the value $min(D)$ is relevant because it represents an error (the agent violates the threshold), in these cases, the algorithm searches for $min(D)$ as close to 0 as possible. In the case when the agent cannot move because of surrounding obstacles and it suspends safely, then the value of f is ∞; that is, the value of f can itself guide the search to skip the obvious cases when the agent is surrounded by obstacles.

5.3 Evolutionary testing

Our testing objective is to assess the robustness of the cleaner agent. In particular, we test only for the capability of the agent to avoid obstacles by using the fitness function f, defined in the previous section. A genetic algorithm (GA) is used to generate test cases that minimize f, that is to find the test cases that lead the agent to breach the threshold ε

(or $f \leq 0$), which is considered as fault.

The experiments were performed on three computers with Intel processors, Core 2 Duo (1.86Ghz), Pentium D (3Gz), and Xeon (4x3Ghz), each has more than 2Gb RAM. Each test case was executed on 5 simulation platforms (i.e., 5 executions per test case) in parallel. The observed data from the platforms was combined to calculate f. In evolutionary testing we choose $\varepsilon = 0.05$, and $w_1 = w_3 = 1/3$.

5.3.1 Experiment 1

In this first experiment, we encode the locations and quantity of waste, obstacles, dustbins and charging stations by four genes: one gene for each kind of object. Their resolutions are chosen as follows:

Kind	Resolution	Max quantity
Waste	12x12	144
Obstacle	8x8	64
Dustbin	2x2	4
Charging station	2x2	4

This experiment was performed on the early beta version of the agent that does not implement the capability to find the shortest path to reach a specific location, and there is no interaction between the two goals: *avoiding-obstacle* and *maintaining-battery*. The latter can inhibit the former while the agent goes to a charging station.

Evolutionary testing is executed with three different configurations: 60, 90 and 120 generations. The best results of all configuration give the optimal value $f = -0.05$ (or the distance to obstacles is 0). This reveals that the agent is faulty, because it hits obstacles. Our testing technique reveals two faults in the implementation of the cleaner agent:

Fault	Description
F1	F1 occurs when the cleaner has two competing goals active at the same time: *maintaining-battery* and *avoiding-obstacles*. The agent favours the goal *maintaining-battery* regardless of the latter goal, so it hits obstacles on the way to a charging station. The value of f corresponding to this fault is very close or equal to the optimal value (-0.05).
F2	F2 is that the agent gets too close to an obstacle before the goal *maintaining-battery* is triggered. The value of f corresponding to this fault is smaller than 0, but far away from the optimal value.

To evaluate the performance of evolutionary testing, we also performed random testing on the cleaner agent. For random testing, test cases were represented in exactly the same way that they were for the evolutionary testing approach, but they were generated entirely randomly. All the settings, such as the resolutions of the objects, values of ε, w_1, w_3, starting point of the agent, and the fitness function f were the same as for evolutionary testing, to ensure fair comparison of results. Three experiments of 60, 90 and 120 random test cases were performed. The evolutionary approach used in this paper is what is known in the literature as a 'steady state genetic algorithm' [20], in which only one new individual is produced at each generation. This means that, at each generation, there is only one new fitness evaluation. We choose settings for random test input generation to ensure that both the random and evolutionary approaches are provided with the same budget of fitness evaluations. In this case, that means choosing the number of random tests to be equal to the number of generations of the evolutionary algorithm. This ensures a fair comparison of the two approaches — evolutionary and random.

Comparing the results obtained from randomly-generated test cases to evolutionary-generated ones, we observe that the fitness values of evolutionary testing are smaller (better) than those of random testing within a similar testing time. In fact, all of the best values of f of evolutionary testing are optimal ($f = -0.05$) while none of the experiments with random testing achieves this value. In addition, the dispersion of distances of evolutionary testing is more compact, and closer to the optimal value than that of random testing. This implies that evolutionary testing generates more challenging test cases to test the cleaner agent than random testing, though both of them can detect the faults.

5.3.2 Experiment 2

The objective of this experiment is to further compare the performance of evolutionary testing to random testing. In this experiment, we fix the locations of 2 charging stations, 2 dustbins, and 6 obstacles (as in Figure 6). The obstacles are placed in the corners so that once the agent goes to these corners, it is difficult for it to get out. In particular, we place three obstacles in the top-right corner, forming a waste–rich potential 'honey pot trap' from which the agent has only one way to get in and out and could drain its battery there. In this experiment only waste is placed randomly in random testing, or with the guidance of the fitness function in ET.

This experiment was performed on a revised version of the cleaner agent. It has the capability to find the shortest trajectory to reach a specific location, avoiding obstacles on the way. Moreover, we change the implementation of the agent to make testing more challenging, by making the first fault *F1* harder to detect. Now in the goal deliberation mechanism of the cleaner agent, the goal *avoiding-obstacles* can inhibit the goal *maintaining-battery* if the battery level is still greater than 5%. The fault has a chance to reveal only the battery level goes below 5%, not 20% like in the previous experiment.

Figure 6: A special scenario to test the cleaner agent

The final results of detecting the two faults *F1* and *F2* of this experiment are described as follows. In three runs with 90, 120, or 200 generations, the evolutionary technique detects both faults; while with comparable numbers of random test cases: 90, 120, 200, the random technique can detect only the easy fault *F2*.

Overall, evolutionary testing, guided by fitness functions

derived from soft-goals, outperforms random testing under the same execution cost and time.

The significance of the test results above is that evolutionary testing, following our approach, tests an agent in a greater range of contexts, thereby accounting for its autonomy to act differently in each. Testing an autonomous agent using a more standard approach can only work if the range of contexts that influence the agent's behaviour is sufficiently limited that the developer can predict them all. However, when considering systems of any substantial complexity, of which a multi-agent system is certainly included, such a limited range is unlikely to occur. We can therefore argue that automated, search-based testing is essential to ensure complex system robustness and, as our tests show, evolutionary testing is an excellent candidate.

6. CONCLUSION AND FUTURE WORK

Autonomous software agents are goal-directed and self-motivated. Their behaviours are seldom determined from external perspectives. As a result, defining test cases to assess the quality of autonomous agents is challenging.

In this paper, we have proposed a systematic way of evaluating the quality of autonomous agents. First, stakeholder requirements are represented as quality measures, and corresponding thresholds are used as testing criteria. Autonomous agents need to meet these criteria in order to be reliable. Fitness functions that represent testing objectives are defined accordingly, and guide our evolutionary test generation technique to generate test cases automatically. The longer the time for evolution, the more challenging the evolved test cases. Thus the autonomous agent is tested more and more extensively.

We developed a simulation of a cleaning agent system to evaluate the proposed approach. The observed results, upon which we report in the paper, demonstrate that evolutionary testing is effective. Indeed, our approach has great potential in evaluating complex software entities like autonomous agents.

For future work, we will consider multiple sets of simultaneous conflicting and competing requirements. For instance, we want to evaluate robustness in terms of maintaining battery and avoiding obstacles. Since each requirement related to autonomy can give rise to a fitness function (or search objective), multiple requirements call for a multi-objective search technique. Fortunately, multi-objective versions of evolutionary algorithms are available to deal with such situations.

7. REFERENCES

[1] J. A. Botía, A. López-Acosta, and A. F. Gómez-Skarmeta. ACLAnalyser: A tool for debugging multi-agent systems. In *Proc. of the European Conference on Artificial Intelligence*, pages 967–968, 2004.

[2] M. E. Bratman. *Intentions, Plans and Practical Reason*. Harvard University Press, 1987.

[3] P. Bresciani, P. Giorgini, F. Giunchiglia, J. Mylopoulos, and A. Perini. Tropos: An Agent-Oriented Software Development Methodology. *Autonomous Agents and Multi-Agent Systems*, 8(3):203–236, July 2004.

[4] O. Bühler and J. Wegener. Evolutionary functional testing. *Computers and Operations Research*, 35(10):3144–3160, 2008.

[5] R. Coelho, U. Kulesza, A. von Staa, and C. Lucena. Unit testing in multi-agent systems using mock agents and aspects. In *Proc. of the 2006 international workshop on Software engineering for large-scale multi-agent systems*, pages 83–90, New York, NY, USA, 2006. ACM Press.

[6] A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20(1-2):3–50, 1993.

[7] O. Dikenelli, R. C. Erdur, and O. Gumus. Seagent: a platform for developing semantic web based multi agent systems. In *Proc. of the fourth international joint conference on Autonomous agents and multiagent systems*, pages 1271–1272, New York, NY, USA, 2005. ACM Press.

[8] E. Gamma and K. Beck. JUnit: A Regression Testing Framework. http://www.junit.org, 2000.

[9] M. P. Georgeff and F. F. Ingrand. Decision-making in an embedded reasoning system. In *Proc. of the International Joint Conferences on Artificial Intelligence*, pages 972–978, 1989.

[10] M. Harman. The current state and future of search based software engineering. In L. Briand and A. Wolf, editors, *Proc. of the IEEE International Conference on Software Engineering, Future of Software Engineering*, pages 342–357, Los Alamitos, California, USA, 2007. IEEE Computer Society Press.

[11] H. Hexmoor, C. Castelfranchi, R. Falcone, M. Luck, M. D'Inverno, S. Munroe, K. S. Barber, I. Gamba, C. E. Martin, S. Braynov, G. Boella, R. Cohen, M. Fleming, S. Franklin, and L. McCauley. *Agent Autonomy*. Springer, 2003.

[12] D. N. Lam and K. S. Barber. *Programming Multi-Agent Systems*, chapter Debugging Agent Behavior in an Implemented Agent System, pages 104–125. Springer Berlin / Heidelberg, 2005.

[13] P. McMinn and M. Holcombe. The state problem for evolutionary testing. In *Proc. of the Genetic and Evolutionary Computation Conference*, pages 2488–2498. Springer-Verlag, 2003.

[14] C. D. Nguyen, A. Perini, and P. Tonella. Automated continuous testing of multi-agent systems. In *The fifth European Workshop on Multi-Agent Systems*, December 2007.

[15] M. Núñez, I. Rodríguez, and F. Rubio. Specification and testing of autonomous agents in e-commerce systems. *Software Testing, Verification and Reliability*, 15(4):211–233, 2005.

[16] A. Pokahr, L. Braubach, and W. Lamersdorf. *Jadex: A BDI Reasoning Engine*, chapter Multi-Agent Programming. Kluwer Book, 2005.

[17] L. F. Rodrigues, G. R. de Carvalho, R. de Barros Paes, and C. J. P. de Lucena. Towards an Integration Test Architecture for Open MAS. In *Proc. of the 1st Workshop on Software Engineering for Agent-oriented Systems / SBES*, 2005.

[18] TILAB. Java agent development framework. http://jade.tilab.com/.

[19] A. M. Tiryaki, S. Öztuna, O. Dikenelli, and R. C. Erdur. Sunit: A unit testing framework for test driven development of multi-agent systems. In *Proc. of the 7th International Workshop on Agent-Oriented Software Engineering*, pages 156–173, 2006.

[20] F. Vavak and T. C. Fogarty. Comparison of steady state and generational genetic algorithms for use in nonstationary environments. In *Proc. of the IEEE International Conference on Evolutionary Computation*, pages 192–195. IEEE Publishing, Inc, 1996.

[21] J. Wegener. *Stochastic Algorithms: Foundations and Applications*, chapter Evolutionary Testing Techniques, pages 82–94. Springer Berlin / Heidelberg, 2005.

[22] D. Weyns, A. Omicini, and J. Odell. Environment as a first class abstraction in multiagent systems. *Autonomous Agents and Multi-Agent Systems*, 14(1):5–30, 2007.